

## Application Notes

# I<sup>2</sup>C Throughput

## 1. TABLE OF CONTENTS

|       |  |   |
|-------|--|---|
| 1.    | TABLE OF CONTENTS .....                            | 2 |
| 2.    | SUMMARY .....                                      | 2 |
| 2.1   | Overview .....                                     | 2 |
| 2.2   | Performance Results .....                          | 2 |
| 3.    | I <sup>2</sup> C PERFORMANCE ANALYSIS.....         | 3 |
| 3.1   | General Considerations.....                        | 3 |
| 3.2   | I <sup>2</sup> C Calculation .....                 | 3 |
| 3.2.1 | I <sup>2</sup> C Time Cost .....                   | 3 |
| 3.2.2 | I <sup>2</sup> C Throughput .....                  | 4 |
| 3.3   | I <sup>2</sup> C on the Messaging Application..... | 4 |
| 3.4   | Conclusion: Maximum Throughput .....               | 5 |
| 4.    | TEST SETUP.....                                    | 6 |

## 2. SUMMARY

### 2.1 Overview

This application note covers I<sup>2</sup>C performance testing of the USEQ\*S devices, from the QFS, QGS and QMS series. The report explains the important data throughput related considerations involved in I<sup>2</sup>C communications design for these sensors. The focus is on the collection of data within the 1ms frame time (i.e. maximum sampling rate). The question is finding the theoretical limit of the I<sup>2</sup>C bus on fast mode plus 1 mbit/s and comparing it to the real-life implementation. The USEQ\*S sensors are compatible with this I<sup>2</sup>C setting.

### 2.2 Performance Results

| Configuration   | Test  | Result            |
|---|---|-------------------|
| I <sup>2</sup> C throughput - theoretical calculation | Reading a 5-channel frame of data per millisecond | <b>4.5 frames</b> |
| I <sup>2</sup> C throughput – API measurement         | Reading a 5-channel frame of data per millisecond | <b>3 frames</b>   |

#### DOCUMENT HISTORY

| Version | Date        | Change Ref. | Change Details |
|---------|-------------|-------------|----------------|
| 01      | 11 JUN 2020 | N/A         | First Release  |
|         |             |             |                |

The above information is believed to be correct but does not purport to be all inclusive and must be used only as a guide.

© KEMET Electronics Corporation • KEMET Tower • One East Broward Boulevard

Fort Lauderdale, FL 33301 USA • 954-766-2800 • www.kemet.com

### 3. I<sup>2</sup>C PERFORMANCE ANALYSIS

#### 3.1 General Considerations

To establish how long it takes for a command to be sent to a USEQ\*S sensor in order to request a full frame of data we need to consider the cost of an operation on the I<sup>2</sup>C bus. In general, a simple transaction will use a certain number of clock cycles in the following manner <sup>1</sup>

2 clocks for the start condition

1 clock for the stop condition

9 clocks for the address

9 clocks per byte sent or received.

The beginning of a transaction between the MCU and the sensor will require  $2+1+9+9 = 21$  clocks.

A typical FIFO\_READ\_ACTIVE operation will recover from the sensor a packet of sampled data of a maximum of 17 bytes: 3 bytes per enabled channel plus 2 bytes for the frame counter.

The fast mode plus I<sup>2</sup>C runs at 1 mbit/s or 1000 kHz at top speed. I<sup>2</sup>C speed varies according to the length of the transaction. A short transaction will not achieve the maximum I<sup>2</sup>C average throughput; a long transaction will get closer to the nominal.

#### 3.2 I<sup>2</sup>C Calculation

##### 3.2.1 I<sup>2</sup>C Time Cost

For this calculation the number of clock periods required for each operation is considered and the number of bytes required for completing the transaction.

| Operation                    | Bytes     | Content          |
|------------------------------|-----------|------------------|
| setup write                  | 1         | sensor address   |
| write command                | 1         | FIFO active code |
| setup read                   | 1         | sensor address   |
| read response                | 17        | data             |
| setup write                  | 1         | sensor address   |
| write command                | 1         | FIFO clear code  |
| setup read                   | 1         | sensor address   |
| read response                | 1         | result code      |
| <b>Bytes per transaction</b> | <b>24</b> |                  |

From the table we see that a complete transaction to read a full packet of data from the sensor requires 24 bytes travelling to and from, between MCU and a USEQ\*S sensor with all active channels collecting data at the maximum rate of 1ms.

<sup>1</sup> Gosh, A. (2012, november 28). *I<sup>2</sup>C communications speed across sensors*. Retrieved from Electrical Engineering: <https://electronics.stackexchange.com/questions/50011/i2c-communication-speed-across-sensors>.

### 3.2.2 I<sup>2</sup>C Throughput

Each byte transmitted or received requires 9 clocks to be delivered, and then we have:

$$9 * 24 = 216 \text{ clocks.}$$

In I<sup>2</sup>C each operation is marked by a beginning and an end using the bus clock. Two clocks are required to start and one to stop.

FIFO active read: start + stop = 3 clocks

FIFO clear : start + stop = 3 clocks

The total numbers of clock cycles required therefore is:  $216 + 6 = 222$  clock cycles.

Our theoretical I<sup>2</sup>C clock runs at 1,000,000 clocks per second, during 1ms we have only 1,000 clocks available, so:

$$1,000 \text{ available} / 222 \text{ required per packet read} = \mathbf{4.50 \text{ frames per millisecond.}}$$

These 4.50 frames contain information of 20 channels spread across 4 USEQ\*S sensors. **Based on this calculation we could expect to read a maximum of 20 channels per millisecond.**

### 3.3 I<sup>2</sup>C on the Messaging Application

Using the messaging application (API) to configure a control board for modules (USEQCSK000000) with four sensors using five channels each, the logic analyser was used to capture the transaction.

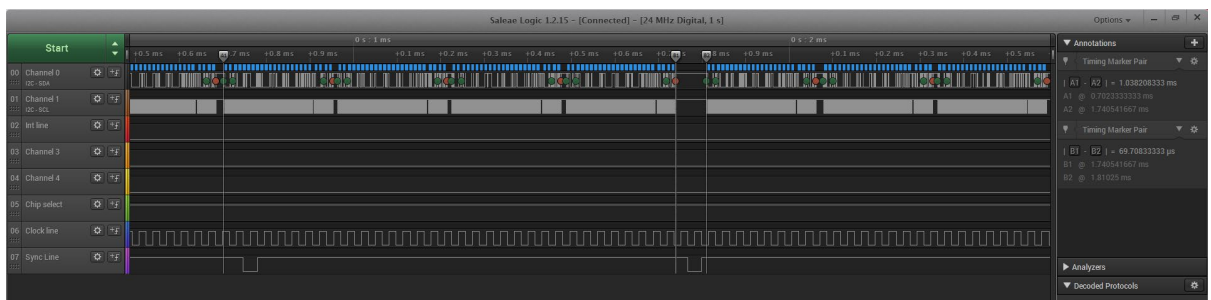


Figure 1 – Logic Analyser Output at Maximum Throughput Captured Data

Figure 1 shows a segment of the I<sup>2</sup>C communication between the USEQ\*S sensor and the MCU. A pair of markers has been set between the beginning and the end of reading a block of four sensors. The logic analyser shows an elapsed time of 1.03 milliseconds to complete the operation. The gap between operations is 0.07 milliseconds. The average clock speed for the I<sup>2</sup>C, reported by the analyser is 888.9 KHz.

However, the GUI application reports a data rate of 464 Hz. When the data is collected into a CSV file, half of the packets of the sampling are lost. Due to the processing overhead that takes place between the firmware and the GUI and the time the next sample data becomes available.

To achieve the best compromise between number of sensors and maximum number of channels sampled, several configurations of number of active channels and number of sensors were tested.

By changing the configuration of the firmware and observing the data rate reported by the USEQ\*S GUI 15 channels could be sampled with the following distribution:

| Sensor | Channels enabled |
|--------|------------------|
| 1      | 5                |
| 2      | 5                |
| 3      | 5                |
| 4      | 0                |

The above information is believed to be correct but does not purport to be all inclusive and must be used only as a guide.

The collected data is shown on the logic analyser below.

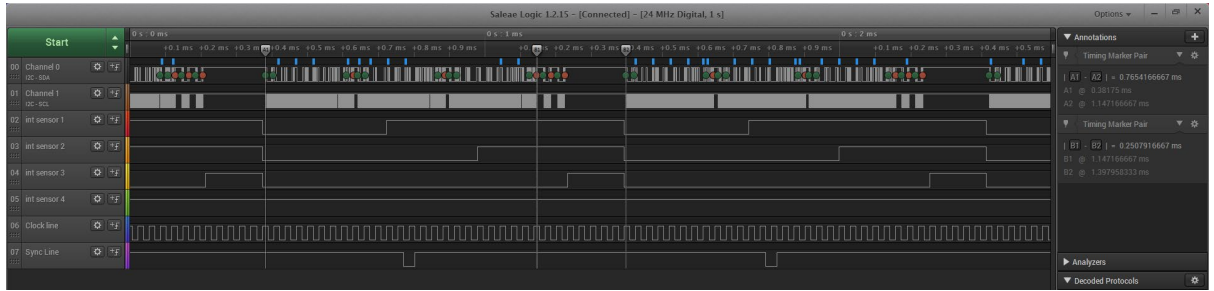


Figure 2 – Logic Capture with no Frames Lost on the GUI

The first marker reports 0.76 milliseconds for the 15 channels read. Two false reads take place that the firmware execute as part of the normal execution loop, when there is no data on the sensors. The last marker shows the gap between reads, when no data is available, which amounts to 0.25 milliseconds.

### 3.4 Conclusion: Maximum Throughput

From the perspective of collecting data with no frames lost and transmitting this data to the existing GUI and the current USEQ\*S API based firmware can record data from **15 channels across 3 sensors**.

If a mechanism to store the collected data in a separate thread is used, then the **theoretical maximum of 20 channels across 4 sensors could be achieved**.

There is an overhead due to normal execution of the firmware that limits the number of channels that can be collected without affecting data collection from the user perspective. How much can this be improved by optimizing the structure of the current firmware could be investigated if required.

### 4. TEST SETUP

Data was collected using Saleae logic analyser, with the I<sup>2</sup>C decoder set to 7-bit addresses <sup>2</sup>. Special modification to the API and the messaging application.

```

606 **@param this_device pointer to the ezPyro data structure
607 **@retval op_result 1 success / 0 error
608 **/
609 static uint8_t CCP_init(ezPyro_sensor_device_type* this_device)
610 {
611     uint8_t op_result = 0, initialization_failed = 0;
612     uint8_t selected_channel = 0, ch_to_activate = 0;
613
614     CCP_INIT_DEFAULT(ccp_defaults);
615     CCP_req_type CCP_data;
616     // active channels value is checked for 1 or 4 channels, if empty defaults to 1
617     switch (this_device->number_of_active_channels)
618     {
619     case 1:
620         ch_to_activate = 1;
621         selected_channel = 2;
622         break;
623     case 4:
624         ch_to_activate = 4;
625         selected_channel = 1;
626         break;
627     default:
628         // other than 1 or 4 the initialization fails.
629         initialization_failed = 1;
630         break;
631     }
632     if (!initialization_failed)
633     {
634         this_device->number_of_active_channels = ch_to_activate;
635         // only active channels are initialized with default data if empty
636         while (ch_to_activate)
637         {
638             CCP_data = this_device->CCP_register(selected_channel);
639             if (CCP_register_empty(CCP_data))
640             {
641                 this_device->CCP_register(selected_channel) = ccp_defaults;
642             }
643             selected_channel =
644             ch_to_activate--;
645             op_result = CCP_write(this_device);
646         }
647         return op_result;
648     }
649     return 0;
650 }
651 /**@brief reads the CCP registers on the ezPyro and loads into this device
652 **@param this_device pointer to the ezPyro data structure
653 **@retval CCP_data
654 **/

```

Figure 3 – CCP\_init() Modification

```

165 this_list[1].CS_Pin = 5;
166 this_list[1].INT_Port = 1;
167 this_list[1].INT_Pin = 6;
168 this_list[1].address = 0x69;
169
170 this_list[2].CS_Port = 2;
171 this_list[2].CS_Pin = 4;
172 this_list[2].INT_Port = 1;
173 this_list[2].INT_Pin = 6;
174 this_list[2].address = 0x68;
175
176 this_list[3].CS_Port = 2;
177 this_list[3].CS_Pin = 3;
178 this_list[3].INT_Port = 1;
179 this_list[3].INT_Pin = 7;
180 this_list[3].address = 0x67;
181
182 //*****COMMIT CONFIGURATION*****//
183 this_list[0].number_of_active_channels = 5;
184 this_list[0].AFE_register = ezPyro_AFE_Master; // MASTER, INT, CLR_ENABLED
185 this_list[0].CCP_register[2] = ezPyro_Ch_Cfg;
186 this_list[0].WUP_register = ezPyro_WUP_Cfg;
187
188 this_list[1].number_of_active_channels = 1;
189 this_list[1].AFE_register = ezPyro_AFE_Slave; // SLAVE, INT, CLR_DISABLED
190 this_list[1].CCP_register[2] = ezPyro_Ch_Cfg;
191 this_list[1].WUP_register = ezPyro_WUP_Cfg;
192
193 this_list[2].number_of_active_channels = 1;
194 this_list[2].AFE_register = ezPyro_AFE_Slave; // SLAVE, INT, CLR_DISABLED
195 this_list[2].CCP_register[2] = ezPyro_Ch_Cfg;
196 this_list[2].WUP_register = ezPyro_WUP_Cfg;
197
198 this_list[3].number_of_active_channels = 1;
199 this_list[3].AFE_register = ezPyro_AFE_Slave; // SLAVE, INT, CLR_DISABLED
200 this_list[3].CCP_register[2] = ezPyro_Ch_Cfg;
201 this_list[3].WUP_register = ezPyro_WUP_Cfg;
202 }
203 /**
204 **@brief Detect ezPyro sensors present in the demo board
205 **@param this_list ezPyro sensor arrays
206 **@param list_size number of sensors in the arrays
207 **/
208 static void ezPyro_sensor_population(ezPyro_devices this_list, uint8_t* list_size)
209 {
210     uint8_t num_of_esPyro = *list_size;
211     uint8_t devices_present = 0;
212     const uint8_t types[] = {0x00, 0x7F, 0x77, 0x67, 0x47};
213     for (uint8_t current = 0; current < num_of_esPyro; ++current)

```

Figure 4 – main.c Modification

<sup>2</sup> Saleae. (2018, January 16). Learn I<sup>2</sup>C - Inter-Integrated Circuit. Retrieved from Saleae.com: <https://support.saleae.com/tutorials/learning-portal/learning-resources/learn-i2c#learn-i-2-c-inter-integrated-circuit>

The above information is believed to be correct but does not purport to be all inclusive and must be used only as a guide.